Fachhochschule
Dortmund
University of Applied Sciences and Arts
Fachbereich Informatik

Algorithmen und Datenstrukturen

VL02 – Komplexität

Unterlagen nur zum persönlichen Gebrauch für Teilnehmer der Lehrveranstaltung „Algorithmen und Datenstrukturen“ an der FH Dortmund!

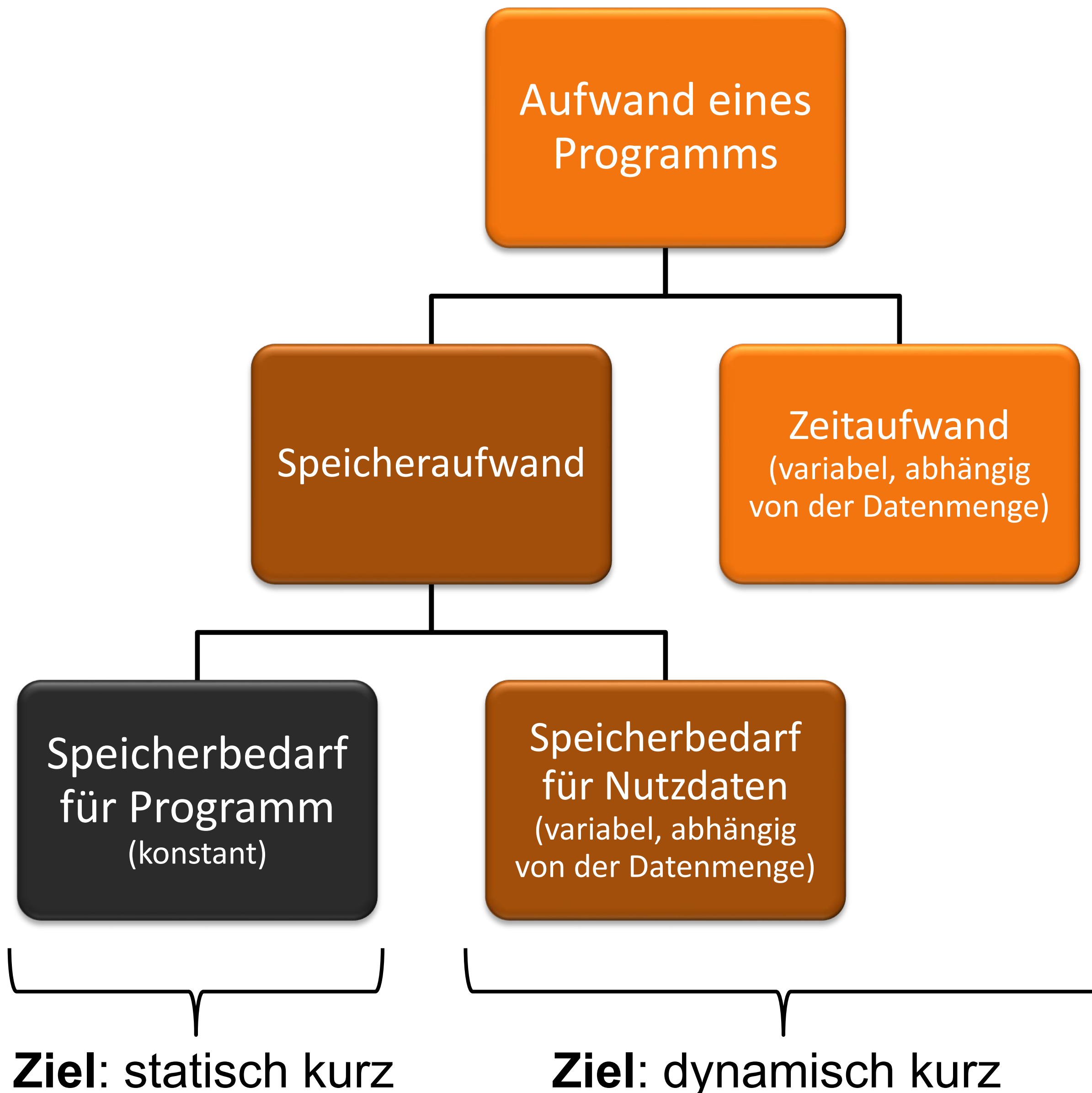
Inhalt

- Was sind Aufwand und Komplexität?
- Asymptotische Zeitkomplexität
- Maximaler Problemumfang

Was sind Aufwand und Komplexität?

Was sind Aufwand und Komplexität?

Ziel ist ein statisch und dynamisches kurzes Programm



- Warum ist der Speicheraufwand in der Praxis eher unwichtig?
  - Der Speicherbedarf für das Programm wächst nicht mit der Datenmenge, sondern bleibt konstant.
  - Der Speicherbedarf für Nutzdaten wächst mit der Datenmenge. Mehr Speicher ist in unserer Welt jedoch billig, aber Rechenzeit bleibt teuer!

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);
  assert(feld.length > 0);

  int max = feld[0];
  for(int a = 1; a < feld.length; a++)
    if (feld[a] > max)
      max = feld[a];

  return max;
}
```

- Speicherbedarf und Laufzeit hängen von der Arraygröße ab
- Wie misst man die Laufzeit auf einer festgelegten Eingabe?
  - Zeitmessung im Programm (siehe Praktikum)
  - Profiler wie `jmc` oder `jvisualvm`

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Zeitmessung für `maximum()` bei 1 Million Elementen:
  - Macbook Pro: 4ms
  - Notebook mit i5-2000: 7ms
  - Raspberry Pi: 40ms

- Warum sind diese Angaben praktisch wertlos?
  - Die Ausführungszeiten sind von vielen Faktoren abhängig:
    - Hardware (unterschiedlicher Prozessor, Takt, Hauptspeichergröße)
    - Unterschiedliche Auslastung (welche Programme liefen im Hintergrund?)
    - Unterschiedliche Betriebssysteme mit unterschiedlichen Eigenschaften
    - Unterschiedliche Programmiersprachen (Java: VM, C++: nativer Code)
    - Temperatur (Drosselung des CPU-Takts bei hoher Wärmeentwicklung)
    - …

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Wir benötigen eine Metrik, die unabhängig von diesen Faktoren ist!

- Annahmen:
  - Ein einzelner Ausführungsschritt (**Taktzyklus**) der CPU benötigt die Zeit c, d.h. wir zählen die Anzahl der Taktzyklen, die ein Programm auf einer Eingabe benötigt und gewichten jeden Schritt mit der Zeit c.
  - Verschiedene Java-Anweisungen benötigen dabei unterschiedlich viele CPU-Befehle und Ausführungsschritte:
    - `for` besteht aus mehreren Operationen wie Zuweisung, Vergleich und Inkrement
    - Division ist langsamer als Addition (z.B. AMD K6: 88 und 1 Zyklen für 32-Bit-Integer)
  - In der Realität hängt die Anzahl Taktzyklen darüber hinaus ab von:
    - Belegung des CPU-Caches durch vorangegangene Operationen („Spectre"-Sicherheitslücken bei Intel- und AMD-CPUs basieren auf diesem Effekt)
    - Konkrete Paarung der CPU-Befehle (CPUs ab Intel Pentium: u- oder v-Pipeline)
    - Dem benutzten CPU-Kern (Apple ARM-CPUs: Hi-Performance oder Hi-Efficiency)
    - …

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Vereinfachtes Modell:
  - Deshalb orientieren wir uns nur sehr grob an der Realität, und legen die Anzahl der Ausführungsschritte für einzelne Operationen fest.

- Ein Ausführungsschritt pro Zuweisung, Rechenoperand, Vergleich, Variablenzugriff und Funktionsaufruf:
  - Je 1 Schritt: `max = 0;` oder Arrayzugriff mit Konstante `feld[0]`
  - Je 2 Schritte: Arrayzugriff über Index `feld[a]`
  - 4 Schritte: `if (feld[a] > max) …`
  - 100 Schritte: `new int[100]`, da jedes Element mit 0 initalisiert wird

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                          0 (keine Auswertung)
  assert(feld.length > 0);                     0 (keine Auswertung)

  int max = feld[0];                           2
  for(int a = 1; a < feld.length; a++)         Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                         4
      max = feld[a];                           3

  return max;                                  1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: Aufsummieren per Einzelschritt-Ausführung

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                          0 (keine Auswertung)
  assert(feld.length > 0);                     0 (keine Auswertung)

  int max = feld[0];                           2
  for(int a = 1; a < feld.length; a++)         Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                         4
      max = feld[a];                           3

  return max;                                  1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: **2**
  (Zugriff auf `feld[0]` und Zuweisung an `max`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                              0 (keine Auswertung)
  assert(feld.length > 0);                         0 (keine Auswertung)

  int max = feld[0];                               2
  for(int a = 1; a < feld.length; a++)             Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                             4
      max = feld[a];                               3

  return max;                                      1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+**1**
  (Initalisierung von `a`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                        0 (keine Auswertung)
  assert(feld.length > 0);                   0 (keine Auswertung)

  int max = feld[0];                         2
  for(int a = 1; a < feld.length; a++)       Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                       4
      max = feld[a];                         3

  return max;                                1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+**3**
  (Vergleich `a < feld.length`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                      0 (keine Auswertung)
  assert(feld.length > 0);                 0 (keine Auswertung)

  int max = feld[0];                       2
  for(int a = 1; a < feld.length; a++)    Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                     4
      max = feld[a];                       3

  return max;                              1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+**4**
  (Vergleich `feld[a] > max`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                    0 (keine Auswertung)
  assert(feld.length > 0);               0 (keine Auswertung)

  int max = feld[0];                     2
  for(int a = 1; a < feld.length; a++)   Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                   4
      max = feld[a];                     3

  return max;                            1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+4+**3**
  (Zugriff auf `feld[a]` und Zuweisung an `max`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                        0 (keine Auswertung)
  assert(feld.length > 0);                   0 (keine Auswertung)

  int max = feld[0];                         2
  for(int a = 1; a < feld.length; a++)       Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                       4
      max = feld[a];                         3

  return max;                                1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+4+3+**1**
  (`a` inkrementieren)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                      0 (keine Auswertung)
  assert(feld.length > 0);                 0 (keine Auswertung)

  int max = feld[0];                       2
  for(int a = 1; a < feld.length; a++)     Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                     4
      max = feld[a];                       3

  return max;                              1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+4+3+1+**3**
(Vergleich `a < feld.length`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                            0 (keine Auswertung)
  assert(feld.length > 0);                       0 (keine Auswertung)

  int max = feld[0];                             2
  for(int a = 1; a < feld.length; a++)           Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                           4
      max = feld[a];                             3

  return max;                                    1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+4+3+1+3+**4**
  (Vergleich `feld[a] > max`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                      0 (keine Auswertung)
  assert(feld.length > 0);                 0 (keine Auswertung)

  int max = feld[0];                       2
  for(int a = 1; a < feld.length; a++)     Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                     4
      max = feld[a];                       3

  return max;                              1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+4+3+1+3+4+**3**
  (Zugriff auf `feld[a]` und Zuweisung an `max`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                         0 (keine Auswertung)
  assert(feld.length > 0);                    0 (keine Auswertung)

  int max = feld[0];                          2
  for(int a = 1; a < feld.length; a++)        Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                        4
      max = feld[a];                          3

  return max;                                 1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+4+3+1+3+4+3+**1**
  (`a` inkrementieren)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                        0 (keine Auswertung)
  assert(feld.length > 0);                   0 (keine Auswertung)

  int max = feld[0];                         2
  for(int a = 1; a < feld.length; a++)       Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                       4
      max = feld[a];                         3

  return max;                                1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+4+3+1+3+4+3+1+**3**
  (Vergleich `a < feld.length`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                          0 (keine Auswertung)
  assert(feld.length > 0);                     0 (keine Auswertung)

  int max = feld[0];                           2
  for(int a = 1; a < feld.length; a++)         Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                         4
      max = feld[a];                           3

  return max;                                  1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+4+3+1+3+4+3+1+3+1 (`return max`)

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                       0 (keine Auswertung)
  assert(feld.length > 0);                  0 (keine Auswertung)

  int max = feld[0];                        2
  for(int a = 1; a < feld.length; a++)      Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                      4
      max = feld[a];                        3

  return max;                               1
}
```

- Gegeben: `feld = { 3, 4, 5 }`
- Ausführungszeit: 2+1+3+4+3+1+3+4+3+1+3+1 = 29 Schritte

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);
  assert(feld.length > 0);

  int max = feld[0];
  for(int a = 1; a < feld.length; a++)
    if (feld[a] > max)
      max = feld[a];

  return max;
}
```

- Laufzeit für `feld = { 3, 4, 5 }` beträgt 29 Schritte
- Laufzeit für `feld = { 3, 3, 3 }` beträgt 23 Schritte
- Für $c = 1\text{ns}$ ergeben sich somit Laufzeiten von 29ns bzw. 23ns

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Wir wollen Algorithmen nicht aufgrund **konkreter** Eingaben bewerten, sondern nur aufgrund des **Problemumfangs**:
  - Die Größe der Eingabedaten wird durch eine ganze Zahl n angegeben; dies ist der Problemumfang.
- Der Zeitaufwand eines Algorithmus wird in Abhängigkeit vom Problemumfang ausgedrückt und als **Zeitkomplexität** des Algorithmus bezeichnet.
- Beispiel `maximum()`:
  - Laufzeit für `feld = { 3, 4, 5 }` beträgt 29 Schritte
  - Laufzeit für `feld = { 3, 3, 3 }` beträgt 23 Schritte
  - Problemumfang in beiden Fällen: $n = 3$

Was sind Aufwand und Komplexität?

Problemumfang

- Offensichtlich kann die tatsächliche Laufzeit für unterschiedliche Eingabedaten trotz identischen Problemumfangs variieren.

- Wir unterscheiden daher zwischen der **minimalen**, **maximalen** und **durchschnittlichen** Komplexität (**best case**, **worst case** und **average case** genannt):
  - Sowohl für den Speicheraufand als auch den Zeitaufwand
  - Falls nicht explizit angegeben, so benutzen wir die maximale Ausführungszeit und den maximalen Speicheraufwand.
  - Die Ausführungszeit und der Speicheraufwand eines Programms P wird häufig auch mit t_P bzw. s_P abgekürzt.

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                          0 (keine Auswertung)
  assert(feld.length > 0);                     0 (keine Auswertung)

  int max = feld[0];                           2
  for(int a = 1; a < feld.length; a++)         Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                         4
      max = feld[a];                           3

  return max;                                  1
}
```

- Die maximale Laufzeit von `maximum()` für ein Feld der Länge n beträgt $7+11(n-1)$:
  - Ohne Schleifendurchlauf: 7 Ausführungsschritte
  - Jeder der n-1 Schleifendurchläufe: bis zu 11 Ausführungsschritte

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);                          0 (keine Auswertung)
  assert(feld.length > 0);                     0 (keine Auswertung)

  int max = feld[0];                           2
  for(int a = 1; a < feld.length; a++)         Initalisierung 1, Iter. 3+1
    if (feld[a] > max)                         4
      max = feld[a];                           3

  return max;                                  1
}
```

- Die maximale Laufzeit von `maximum()` für ein Feld der Länge n beträgt $7+11(n-1)$:
  - Ohne Schleifendurchlauf: 7 Ausführungsschritte
  - Jeder der n-1 Schleifendurchläufe: bis zu 11 Ausführungsschritte

Was sind Aufwand und Komplexität?

Ermittlung des Aufwands

- Beispiel: gleiche Elemente in einem Array

```
public static boolean twins(int[] feld)
{
  assert(feld!=null);
  assert(feld.length > 0);

  for(int a = 0; a < feld.length; a++)
    for(int b = 0; b < feld.length; b++)
      if ((a != b) & (feld[a] == feld[b]))
        return true;

  return false;
}
```

- Aufgabe: was ist die maximale Laufzeit von `twins()` für ein Feld der Länge n?

$$5+n(8+13n) = 5+8n+13n^2$$

Was sind Aufwand und Komplexität?

Effizienz groß genug?

- Aufwand von Algorithmen für große n:
  - Unterschiedliche Anforderungen:
    - In harter Echtzeit (ms oder schneller)
    - In wenigen Sekunden
    - …über Nacht
  - Sind Algorithmen für gegebene Problemgröße effizient genug?

		Problemgröße 1 Million		Problemgröße 1 Milliarde	
		$t(n) \leq c*n$	$t(n) \leq c*n^2$	$t(n) \leq c*n$	$t(n) \leq c*n^2$
Zeit für eine Operation c	10^{-12}s	µs	Sekunden	ms	Wochen
	10^{-9}s	ms	Minuten	Sekunden	Jahrzehnte
	10^{-6}s	Sekunden	Wochen	Stunden	Niemals

1 Stunde	2 Wochen	100 Jahre = 10 Jahrzehnte	Niemals (≥ 100 Jahrtausende)
$3{,}6*10^3$s	$1{,}2*10^6$s	$3{,}15*10^9$s	$\geq 3{,}15*10^{12}$s

Asymptotische Zeitkomplexität

Asymptotische Zeitkomplexität

O-Notation

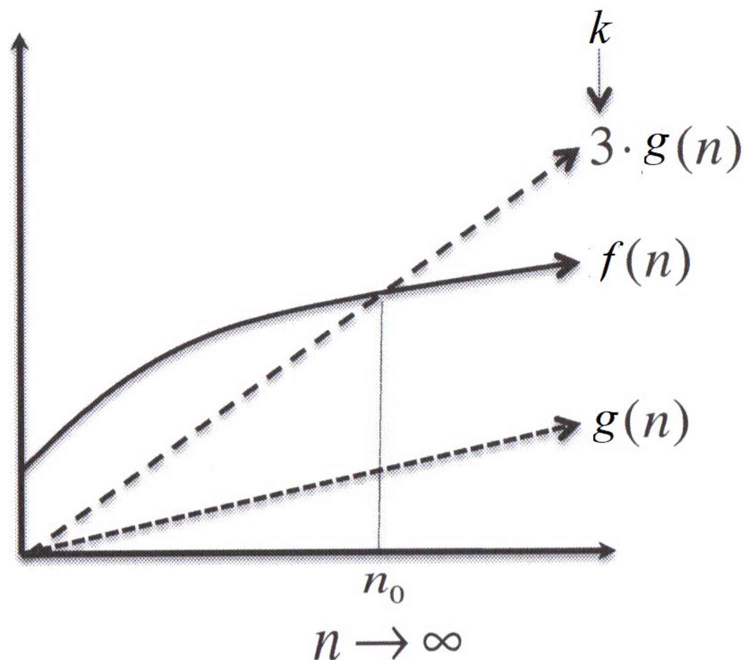
- Die Zeitkomplexität t eines Programms ist eine Funktion, die jeder Problemgröße n eine Laufzeit $t(n)$ zuordnet, d.h. $t: \mathbb{N} \longrightarrow \mathbb{N}$.

- **O-Notation** (Landau-Symbole):
  - Seien f und g zwei Funktionen auf den natürlichen Zahlen, also $f, g: \mathbb{N} \longrightarrow \mathbb{N}$
  - Die Funktion f wächst asymptotisch für $n \rightarrow \infty$ nicht stärker als g, kurz $f = O(g)$, falls es Konstanten k und n_0 gibt mit $f(n) \leq k * g(n)$ für alle $n \geq n_0$.
  - $O(g)$ ist die Menge aller Funktionen f mit $f = O(g)$. Formal korrekt müsste man $f \in O(g)$ schreiben, jedoch hat sich $f = O(g)$ historisch durchgesetzt.

Asymptotische Zeitkomplexität

O-Notation

- O-Notation in Wort und Bild:
  - Seien f und g zwei Funktionen auf den natürlichen Zahlen, also $f, g\colon \mathbb{N} \longrightarrow \mathbb{N}$
  - Die Funktion f wächst asymptotisch für $n \to \infty$ nicht stärker als g, kurz $f = O(g)$, falls es Konstanten k und n_0 gibt mit $f(n) \leq k * g(n)$ für alle $n \geq n_0$.



in Anlehnung an: Roughgarden, 2017

Asymptotische Zeitkomplexität

O-Notation

- Beispiel1: $f(n) = 14n+5, g(n)=n$

  Es gilt: $14n+5 \leq 14n + 5n = 19n$, für alle $n \geq 1$

  d.h. $k = 19$ und $n_0 = 1$

  $\Rightarrow \; f(n)=O(g(n))=O(n)$

- Beispiel2: $f(n) = n^3+4n^2, g(n)= n^3$

  Es gilt: $n^3+4n^2 \leq n^3+4n^3 = 5n^3$, für alle $n \geq 1$

  d.h. $k =5$ und $n_0 = 1$

  $\Rightarrow \; f(n)=O(g(n))=O(n^3)$

Asymptotische Zeitkomplexität

O-Notation

- Zur Beantwortung der Frage, ob für zwei Funktionen f und g gilt $f=O(g)$ kann man versuchen den Grenzwert $\lim_{n\to\infty}\frac{f(n)}{g(n)}$ zu verwenden (siehe auch Mathematik-Vorlesungen):
  - Existiert ein Grenzwert, so gilt $f=O(g)$.
  - Divergiert $\frac{f(n)}{g(n)}$ bestimmt, d.h. $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$, dann gilt $f\neq O(g)$.
  - Divergiert $\frac{f(n)}{g(n)}$ unbestimmt, so muss man prüfen, ob es eine obere Schranke gibt.

Asymptotische Zeitkomplexität

O-Notation

- Beispiel 1: gilt $14n^2+8n+12=O(n^3)$?
  - $$\lim_{n\to\infty}\frac{14n^2+8n+12}{n^3}=\lim_{n\to\infty}\frac{14/n+8/n^2+12/n^3}{1}=0$$
  - Es existiert ein Grenzwert (nämlich 0), so dass die Aussage korrekt ist.

- Beispiel 2: gilt $2n^3+8n=O(n^2)$?
  - $$\lim_{n\to\infty}\frac{2n^3+8n}{n^2}=\lim_{n\to\infty}\frac{2n+8/n}{1}=\infty$$
  - Die Folge divergiert bestimmt, somit ist die Aussage falsch.

Asymptotische Zeitkomplexität

O-Notation

- Rechenregeln: es gelte $f=O(g)$ und $f'=O(g')$. Dann gilt auch:
  - $O(k*g+k') = O(g)$ für alle Konstanten $k,k'>0$
  - $f+f' = O(g+g') = O(\max(g, g'))$
  - Falls $g, g' > 0$ so gilt auch $f*f'=O(g*g')$

Asymptotische Zeitkomplexität

O-Notation

- Aus den Rechenregeln lässt sich folgende Faustregel ableiten, um für eine Funktion f eine passende Funktion g zu finden, so dass f=O(g) gilt:
- **„Asymptotic Notation in Seven Words:**
  - suppress constant factors and lower-order terms“

(Quelle: Roughgarden, 2017)

auf deutsch:

- “Vernachlässige konstante Faktoren und Terme (genauer Summanden) niederer Ordnung (d.h. geringeren Funktionswachstums)“

- Beispiel: $f(n) = 14n^2+8n+12$

Konstanter Faktor Terme niederer Ordnung

Es gilt also: $f(n) = O(n^2)$

Asymptotische Zeitkomplexität

O-Notation

- Die Zeitkomplexität t eines Programms ist eine Funktion, die jeder Problemgröße n eine Laufzeit $t(n)$ zuordnet, d.h. $t: \mathbb{N} \longrightarrow \mathbb{N}$.
  - Gilt für die Zeitkomplexität t eines Algorithmus $t(n)=O(f(n))$, so sprechen wir auch davon, dass der Algorithmus eine Laufzeit oder Zeitkomplexität $O(f(n))$ besitzt.
  - Gilt zusätzlich die Umkehrung $f(n)=O(t(n))$, d.h. $O(t)=O(f)$, so sprechen wir auch davon, dass die Zeitkomplexität des Algorithmus (oder kurz: der Algorithmus) die **Ordnung** $O(f(n))$ hat.

- Bei der Analyse eines Algorithmus wollen wir immer die **genaue Ordnung** der Laufzeit ermitteln!
  - Beispielsweise gilt für fast alle Algorithmen $t(n)=O(2^n)$.
  - Diese Aussage ist dadurch allerdings ziemlich wertlos.
  - Analog dazu wird wie bei einer Personenbeschreibung auf Aussagen wie „Hat eine Nase“ verzichtet, da dies ohnehin fast immer zutrifft.

Asymptotische Zeitkomplexität

O-Notation

- Beispiel: Maximum in einem Array

```
public static int maximum(int[] feld)
{
  assert(feld!=null);
  assert(feld.length > 0);

  int max = feld[0];
  for(int a = 1; a < feld.length; a++)
    if (feld[a] > max)
      max = feld[a];

  return max;
}
```

- Für `maximum()` gilt $t(n)=7+11(n-1)$ (siehe Folien 26 und 27)
- Die Ordnung von `maximum()` ist also $O(n)$. Beweis:

$$\lim_{n\to\infty}\frac{7+11(n-1)}{n}=\lim_{n\to\infty}\frac{11n-4}{n}=11\wedge\lim_{n\to\infty}\frac{n}{7+11(n-1)}=\lim_{n\to\infty}\frac{n}{11n-4}=\frac{1}{11}$$

Asymptotische Zeitkomplexität

O-Notation

- Beispiel: gleiche Elemente in einem Array

```
public static boolean twins(int[] feld)
{
  assert(feld!=null);
  assert(feld.length > 0);

  for(int a = 0; a < feld.length; a++)
    for(int b = 0; b < feld.length; b++)
      if ((a != b) & (feld[a] == feld[b]))
        return true;

  return false;
}
```

- Für `twins()` gilt $t(n)=5+8n+13n^2$ (siehe Folie 28)
- Die Ordnung von `twins()` ist $O(n^2)$.

Asymptotische Zeitkomplexität

O-Notation

- Aufgabe: welcher Ordnung gehört die folgende Methode an?

```
public static void methode(int n)
{
  for(int a = 0; a < n*n; a++)
    for(int b = 0; b < n*n*n; b++)
      tuwas();                    // Wieviele Aufrufe?
}
```

Die Ordnung ist $O(n^5)$.

Asymptotische Zeitkomplexität

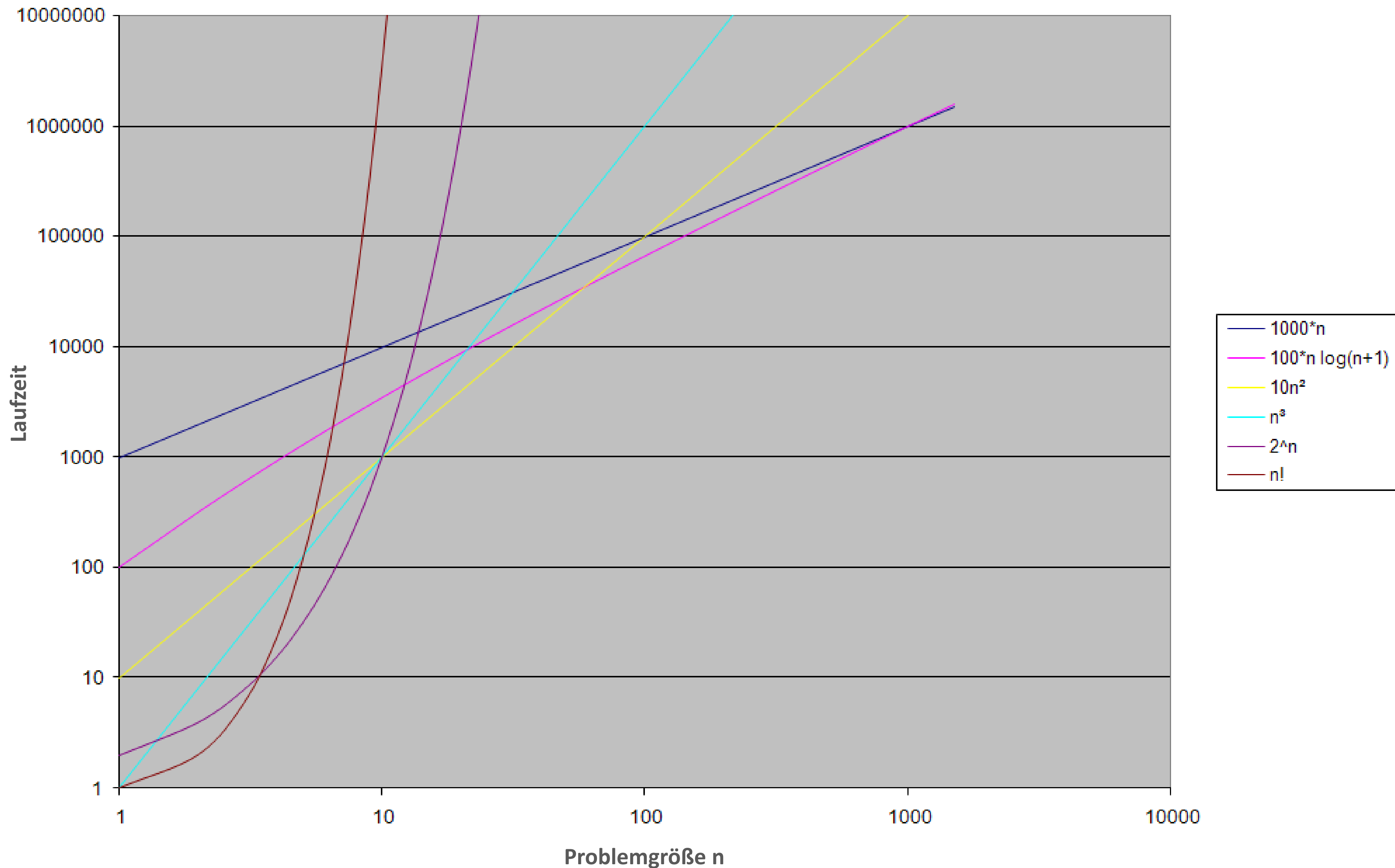
Typische Ordnungen

Ordnung	Name
$O(1)=O(k)$	Konstant
$O(\log n)$	Logarithmisch
$O(n)$	Linear
$O(n^2)$	Quadratisch
$O(n^3)$	Kubisch
$O(n^k)$	Polynomiell
$O(2^{kn})$	Exponentiell ☠💣

- Es gilt: $O(1) \subsetneq O(\log n) \subsetneq O(n) \subsetneq O(n^2) \subsetneq O(n^3) \subsetneq O(2^{kn})$

Asymptotische Zeitkomplexität

Welcher Algorithmus ist effizienter ?



Maximaler Problemumfang

Maximaler Problemumfang

Berechnung des maximalen Problemumfanges

- Gegeben:
  - Ausführungsgeschwindigkeit, z.B. $c=10^{-6}s$
  - Algorithmus der Ordnung $O(n)$

- Gesucht:
  - Maximaler Umfang eines Problems, der mit Algorithmen der Ordnung $O(n)$ bearbeitet werden kann

- Ergebnis:
  - Maximaler Problemumfang n bei 1 Sekunde: $n=10^6$
  - Maximaler Problemumfang n bei 1 Minute: $n=6*10^7$
  - Maximaler Problemumfang n bei 1 Stunde: $n=3,6*10^9$

Maximaler Problemumfang

Problemumfang für verschiedene Ordnungen

- Ausführungsgeschwindigkeit $c=10^{-6}s$:

Zeitkomplexität	Maximaler Problemumfang			Beispiel
	1 Sekunde	1 Minute	1 Stunde	
$O(n)$	1.000.000	60.000.000	$3,6*10^9$	`maximum()`
$O(n \log_2 n)$	62.746	12.800.000	$0,313*10^9$	Heapsort
$O(n^2)$	1000	7745	60.000	`twins()`
$O(n^3)$	100	391	1.532	Gaußscher Alg.
$O(2^n)$	19	25	31	Türme von Hanoi
$O(n!)$	9	11	12	Wegoptimierung

Maximaler Problemumfang

Problemumfang für verschiedene Ordnungen

- Zunahme des maximalen Problemumfangs bei Verzehnfachung der Geschwindigkeit ($c=10^{-7}s$):

Zeitkomplexität	Maximaler Problemumfang vor der Verzehnfachung	Maximaler Problemumfang nach der Verzehnfachung
$O(n)$	n	$n'=10*n$
$O(n \log_2 n)$	n	$n'\approx 10*n$
$O(n^2)$	n	$n'=3{,}16*n$
$O(n^3)$	n	$n'=2{,}15*n$
$O(2^n)$ ☠💣	n	$n'=n+3{,}3$
$O(n!)$ ☠💣	n	$n'=n+\frac{1}{\log_{10}(n+1)}$

- Dies ist möglicherweise die wichtigste Tabelle Ihres gesamten Studiums! Was fällt auf?
  - Bei expontiellen Algorithmen wächst der Problemumfang um keinen Faktor mehr, sondern nur noch um einen Summanden!

Maximaler Problemumfang

Zusammenhang Zeit – Problemumfang

- Fazit:
  - Hat ein Algorithmus eine große Zeitkomplexität (insbesondere schlechter als polynomiell), so kann selbst eine starke Erhöhung der Rechenleistung den maximalen Problemumfang kaum steigern.
  - Durch eine Verbesserung der Zeitkomplexität erhält man meistens eine bessere Effizienz als durch Steigerung der Rechenleistung.
  - **Aber:** ist der Problemumfang klein, so entscheiden konstante Terme und Faktoren über den effizientesten Algorithmus. So ist beispielsweise für $n \leq 4$ ein Algorithmus mit der Zeitkomplexität $f(n) = n!$ besser als ein Algorithmus mit $g(n) = 10 * n$:

n	**1**	**2**	**3**	**4**	**5**	**6**
n!	1	2	6	24	120	720
10n	10	20	30	40	50	60

Lernziele

- Sie kennen die Aufwandskriterien für Algorithmen und Programme
- Sie kennen die verschiedenen Formen von Zeit- und Speicherkomplexität
- Sie können den Begriff Problemgröße erläutern
- Sie können die Zeitkomplexität von einfachen Algorithmen und Programmen bestimmen und deren Ordnung in O-Notation angeben
- Sie kennen häufig vorkommende Größenordnungen und können diese einordnen

Literatur

- Quellen:
  - Balzert, H.: Lehrbuch Grundlagen der Informatik (LE 16: Aufwand von Algorithmen)
  - Saake, G.: Algorithmen und Datenstrukturen (Kapitel 7.3: Komplexität)
  - Roughgarden, T.: Algorithms Illuminated, Part 1: The Basics (Chapter 2: Asymptotic Notation)